Trading Bandwidth for Latency: Managing Continuations Through a Carpet Bag Cache [*]

Richard C. Murphy and Peter M. Kogge
Computer Science and Engineering Department
University of Notre Dame
{rcm,kogge}@cse.nd.edu

Abstract

Processing-In-Memory (PIM) circumvents the von Neumann bottleneck by combining logic and memory (typically DRAM) on a single die. This work examines the performance of a mobile thread execution model in which threads traverse the system's address space in search of their required data over a massively parallel PIM array targeted at petaflop performance. This model is enabled through the use of a *carpetbag cache* which travels with the thread and provides data from the node previously visited. In this way, the latency of traveling to a node already visited is avoided by paying the additional bandwidth and packaging costs associated with moving the cache. Furthermore, it is shown that thrashing between two nodes will generally occur without the use of the cache. Each of the simulations conducted in this work was conducted under stress. By using the Data Intensive Systems (DIS) benchmark suite, the model was subjected to "worst case" memory access patterns, and ultimately still proved highly successful. Additionally, it is shown that this model caters to the positive aspects of a PIM – specifically, the very fast local memory access available.

1 Introduction and Motivation

Processing-in-Memory (PIM)[11, 10, 4] (also known as Intelligent RAM [19], embedded RAM, or merged logic and memory) systems exploit the tremendous amounts of memory bandwidth available for intra-chip communication, and therefore circumvent the von Neumann bottleneck, by placing logic and memory (typically DRAM) on the same die. This technology allows for the construction of highly distributed systems, but with a very large latency gap between high speed local memory macro accesses and remote accesses. The construction of high performance systems incorporating PIMs must successfully exploit the bandwidth available for on-chip accesses while simultaneously tolerating very long remote access latencies. Multithreading, similar to that used in the Tera[1], seems the natural method for tolerating remote accesses, however, such a model does not inherently take advantage of the relatively large amount of quickly accessed memory available on a PIM node. In fact, the Tera generally requires about the same amount of persistent state as available in the L1 cache of a modern microprocessor[1], and a typical PIM node is likely to have 3 to 4 orders of magnitude more memory available. Furthermore, very large PIM arrays consisting of up to one million nodes and targeted at petaflop performance require tremendous numbers of threads to tolerate remote access latencies. The problem is compounded when one takes into account the difficulty in translating the name of remote objects or pages which are in motion. Traditional techniques which may scale to O(1000) nodes are generally unsuitable for arrays of this magnitude. Furthermore, those techniques generally do not address the problems encountered by data intensive algorithms (ie, those with large datasets that exhibit low reuse), nor do they fit well into a PIM architecture which must exploit the high speed/high bandwidth local DRAM access to be successful over non-PIM implementations.

This paper examines the *carpetbag cache* protocol to support the movement of a thread from one node to another. This process consists of the packaging, transmission, and continuation of a thread on another node when a remote memory access is generated. It allows for static data placement in which computation pursues data throughout the system. Rather than having a remote memory access cause a fetch, the computation is packaged and sent to the node upon which the remote data resides. It will be shown that this model is highly effective at trading

[*]The work described here was funded in part by the University of Notre Dame, in part by the HTMT project run by the Jet Propulsion Propulsion Laboratory, California Institute of Technology and sponsored by the Defense Advanced Research Projects Agency (DARPA) and the National Security Agency (NSA) through an agreement with the National Aeronautics and Space Administration, and in part by DARPA and Rome Laboratory, Air Force Material Command, USAF, under cooperative agreement number F30602-98-2-0180 as part of the Data IntensiVe Architecture (DIVA) Project under the Data Intensive Systems (DIS) program.

bandwidth (in terms of shipping a large continuation) for latency (in terms of reducing the **number of communication events** as well as emphasizing one way communication over round trip accesses). Furthermore, it will be shown that threads are capable of very long run lengths on a node before moving, thereby emphasizing the fast local memory access available to a PIM node.

This paper is organized as follows: Section 2 reviews PIM architecture in general as well as the specific architecture targeted for the carpetbag cache. Section 3 examines the carpetbag cache architecture. Section 4 discusses the simulation methodology. Section 5 details the two Data Intensive Systems (DIS) benchmarks used during experimentation. And, finally, Sections 6 and 7 present the results of experimentations and the associated conclusions.

2 PIM Architecture

Modern processors require that tremendous amounts of data be provided by the system's memory hierarchy. This demand is becoming increasingly difficult to meet. The core problem, known as the *von Neumann Bottleneck* relates to the separate development of processing and memory technologies, and the different emphasis placed on each. Processors, built around logic fabrication processes which emphasize fast switching, generally follow Moore's law, while memories emphasize high density but relatively low data retrieval rates. The interconnection mechanism between the two is a narrow bus which cannot be greatly expanded due to the physical limit on the number of available pins and high capacitance of inter-chip communication.

Recent developments in VLSI technology, such as the trench capacitor compatible with logic processes created at IBM, now allow for fabrication facilities which offer both high performance logic and high density DRAM on the same die. These PIMs further allow for the creation of much higher bandwidth interconnection between local memory macros and logic since it all occurs on chip.

Several proposals exist which attempt to fully utilize the potential of these fabrication processes. The IRAM project[19] at Berkeley seeks to place a general purpose core with vector capabilities along with DRAM onto a die for embedded applications. Cellular phones, PDAs, and other devices requiring processing power and relatively small amounts of memory could benefit tremendously from this type of system, even if one only considers the potential advantages in power consumption. Others, such as members of the Galileo group[5] at the University of Wisconsin see PIM as having tremendous potential in standard workstations where the on chip memory macros would become all or part of the memory hierarchy. More recently, the Stanford Smart Memories project[14] began exploring the construction of single chip systems capable of supporting a diverse set of system models.

The DIVA project [10] is currently investigating system and chip level implementations for PIM arrays functioning as part of the memory hierarchy in a standard workstation. Finally, the HTMT[20, 12] project is a multi-institutional effort to construct a machine capable of reaching a petaflop or above in which a large part of the memory hierarchy consists of PIMs. This portion of the memory hierarchy is a huge, two-level, multi-threaded array.

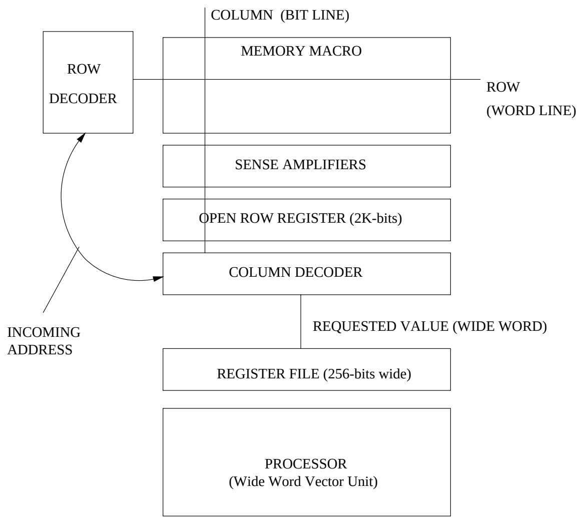

COLUMN (BIT LINE)
MEMORY MACRO
ROW DECODER
ROW (WORD LINE)
SENSE AMPLIFIERS
OPEN ROW REGISTER (2K-bits)
COLUMN DECODER
INCOMING ADDRESS
REQUESTED VALUE (WIDE WORD)
REGISTER FILE (256-bits wide)
PROCESSOR (Wide Word Vector Unit)

Figure 1: Typical PIM Memory Layout

Figure 1 show a typical single node PIM layout. In the case of the target At the Sense Amp Processor (ASAP) Architecture[17], a wide word processor (capable of operating on 256 bit vectors in 8, 16, or 32 bit chunks) is tightly coupled with a set of memory macros. For the purposes of simulation, it is assumed that the memory macro provides 2 k-bits of data per operation through a single open row register. The ASAP's register file then accesses that data in 256 bit chunks as if from an 8 by 256 bit register file. Thus, while a random read from memory will cause a DRAM access, a read contained in the current open row does not incur that penalty (because it is simply a register transfer operation).

It should be noted that, unlike other proposed PIM implementations, the ASAP is designed to be a simple vector machine tightly coupled to the memory macro. Given this tight coupling, it is critical to view a successful memory management scheme in terms of exploiting fast local memory accesses. The simplicity of the processor allows for more of the surface area of the chip can be devoted to memory. Furthermore, a model of latency toleration (through multi-threading) contributes significantly to this notion. This not only takes fuller advantage of the high speed local DRAM access times, but helps in fabrication yield (in that memory defects are more easily addressed than logic defects). Furthermore, chips will contain multiple nodes (processors and associated memory macros) connected through a high speed interconnection network.

The array of PIMs simulated is assumed to be homogeneous. For the purposes of this paper, no particular interconnection topology is assume (rather, communication events are merely counted). Experimentation over various topologies can be found in [15]. In actuality, a PIM array is likely to be heterogeneous (potentially consisting of PIMs of different types – SRAM and DRAM – and different sizes), and the interconnection network hierarchical. Multiple nodes will be present on a chip, facilitating significantly faster on-chip communications mechanisms versus off chip communication mechanisms. Additionally, since PIM systems may be part of a larger memory hierarchy, additional non-PIM processing resources or memory may be available.

PIMs, in our model, communicate through the use of *parcels*, which are akin to Active Messages [22], are messages possessing intrinsic meaning directed at named objects. Rather than merely serving as a repository for data, parcels carry distinct high level commands and some of the arguments necessary to fulfill those commands. Low level parcels (which may be handled entirely by hardware) may contain simple memory requests such as: "access the value X and return it to node K." Higher level parcels are more complicated and may take the form "resume execution of procedure Y with the following partially computed result and return the answer to node L." Thus, it should be assumed that parcels can perform both communication and computation, and may be invoked by the user, runtime system, or hardware.

In terms of latency and bandwidth, latency is represented in the number of transactions generated over the network. Once such a transaction is generated, difference in cost between a long transaction and a short transaction is considered to be negligible. This is characteristic of a high bandwidth interconnection network connecting a very large number of nodes.

3 Carpetbag Cache Architecture

The carpetbag cache supports the movement of a thread from one node to another. In the Mobile Thread Model[15, 16], a thread executes on a given node until it generates a remote memory access. That access causes the thread to be packaged and moved to the node containing the data. Experimentation has shown that given a simple data placement scheme (in which data is divided into large contiguous chunks – see Section 4)) if the thread moves every time a remote access is encountered it will often thrash between two nodes[15, 16]. This thrashing occurs because when the request for remote memory occurs a small amount of data from the node upon which the thread is currently executing is still needed. The carpetbag cache is designed to capture this data.

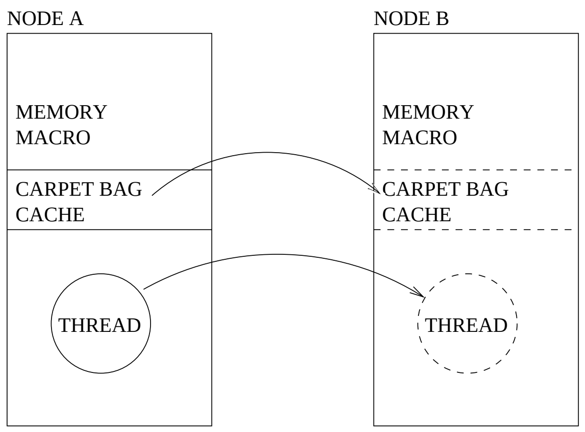


Figure 2: Thread Movement using a Carpetbag Cache

Figure 2 shows the movement of a thread from Node A to Node B. The parcel representing the thread contains the code and stack needed by the thread, its register set, and associated carpetbag cache. The contents of the cache represent data from the current node (in this case A) which is likely to be useful on the next node (B). This data is gathered using a standard caching mechanism. The simulated mechanism is a fully associative 256 bit word data cache with a true LRU replacement policy.

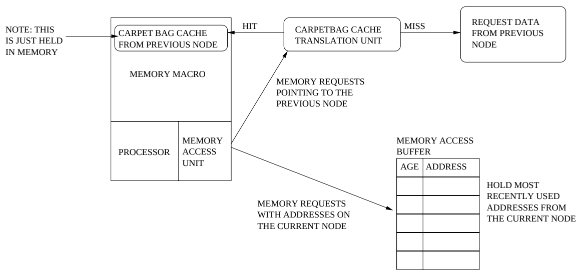

NOTE: THIS IS JUST HELD IN MEMORY
CARPET BAG CACHE FROM PREVIOUS NODE
MEMORY MACRO
HIT
CARPETBAG CACHE TRANSLATION UNIT
MISS
REQUEST DATA FROM PREVIOUS NODE
MEMORY REQUESTS POINTING TO THE PREVIOUS NODE
PROCESSOR
MEMORY ACCESS UNIT
MEMORY ACCESS BUFFER
AGE
ADDRESS
HOLD MOST RECENTLY USED ADDRESSES FROM THE CURRENT NODE
MEMORY REQUESTS WITH ADDRESSES ON THE CURRENT NODE

Figure 3: Carpetbag Cache Construction

Figure 3 shows the mechanism used to construct and utilize a carpetbag cache. The memory macro on each node contains the data used by the carpetbag cache (in 256 bit words). A fully associative translation unit is used to determine where remote accesses pointing to the previous node reside (in the carpetbag cache, or on the previous node). This translation unit could, in fact, be implemented in software if references to the previous node caused a trap into the operating system. For the purposes of this simulation, however, it is assumed that the translation information is immediately available.

Given the nature of the ASAP vector processor, the cache could easily be implemented as a balanced oct tree. The most highly reused words can be held in an actual cache, while the rest can easily be located in the tree. When compared to potentially performing a remote memory access, searching a relatively small 8-ary tree proves relatively painless. Furthermore, as the system relies upon multi-threading, the only overhead associated with the search is the search itself.

The carpetbag cache for the next node is constructed using a Memory Access Buffer. All requests to the local node are placed into this buffer in true LRU order (though other schemes more easily implemented in silicon could be used). When the thread is packaged for movement to the next node, the runtime system can scan the buffer and place any appropriate words into the parcel for transmission. Again, this could be done entirely in hardware, and for the purposes of simulation it is assumed to occur very quickly. In the event that a memory request which references the previous node is generated but misses the carpetbag cache, a read request occurs and the cache records a miss.

This arrangement leads to a potential synchronization problem in that words contained in the carpetbag cache of another thread may be requested on a different node. Though a complex coherency protocol could be used in future work, it is assumed that the machine under examination contains a full/empty bit for each 256 bit word. When a word is placed in the carpetbag cache for a given thread, the word is marked empty in the local node's memory macro. Any thread attempting to access that word blocks and is placed on a queue until the word is marked full. When a thread moves from one node to another, the dirty words in its carpetbag cache are copied back to the previous node so that those words may be marked full, the new cache is then constructed from data on the current node. In this way, the carpetbag cache **only** contains data from the thread's previous node. A more general system (discussed briefly in [15]) adds significant additional synchronization complexity.

Figure 4 shows the procedure used to translate a memory access and fetch the appropriate location.

This particular scheme may appear inflexible in the face of read-read sharing. The data intensive benchmarks presented in this work (see Section 5) generally exhibit low degrees of read-read sharing (this is precisely why they are so difficult for most modern architectures). There are generally two strategies for supporting read-read sharing in the mobile thread model. First, the compiler (or user, or runtime system) can mark words in memory as being read-read shared. This would allow the memory system to freely copy them, however traditional locks would have to be used to perform updates. (More generally, the data could also be replicated across nodes and marked as unavailable to the carpetbag cache.) The second approach is to rely upon the multi-threaded execution used by the machine to support sharing. In the presence of sufficient concurrency, having threads blocked on read-read shares is not particularly difficult for the system to handle. This is especially true given that the carpetbag cache in this model **only holds data from the previous node visited.** Simulation shows that threads move reasonably often, and, upon movement, the updated contents of the carpetbag cache are returned to the previous node.

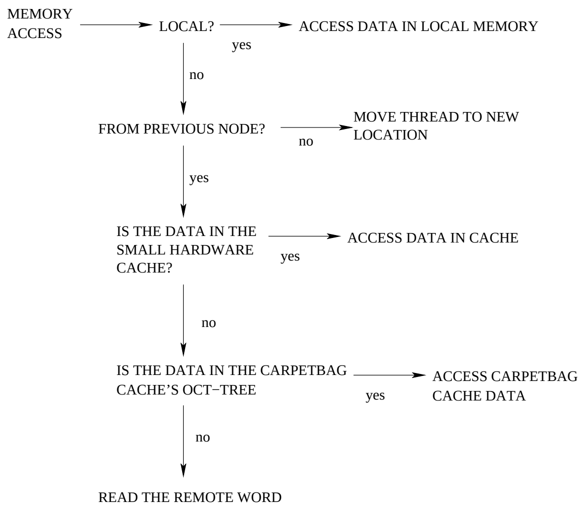

MEMORY ACCESS
LOCAL?
yes
ACCESS DATA IN LOCAL MEMORY
no
FROM PREVIOUS NODE?
no
MOVE THREAD TO NEW LOCATION
yes
IS THE DATA IN THE SMALL HARDWARE CACHE?
yes
ACCESS DATA IN CACHE
no
IS THE DATA IN THE CARPETBAG CACHE'S OCT−TREE
yes
ACCESS CARPETBAG CACHE DATA
no
READ THE REMOTE WORD

Figure 4: Memory Access Procedure

Ultimately, this model emphasizes the following:

- Page tables or other data structures managing the translation of names become small.
- Static data placement significantly reduces the synchronization involved in updating distributed versions of those structures.
- The physical location of a given computation need not be tracked at all. Threads can freely roam the system without causing the update of complicated, distributed data structures. Specifically, if various threads communicate through shared memory, they need not know the physical node upon which the thread with which they are communicating resides, only the location of the shared memory.
- Programming models can emphasize moving to a given node, exhausting the data present, and moving on. Simple mechanisms for delivering such data can easily be provided by the runtime system.

- No round trip communication is necessary since the thread can move to the data rather than requesting data which must then be returned. This eliminated one high latency penalty upon each movement.

The potential disadvantages are:

- Load balancing may be difficult, especially if data placement relies upon highly shared data structures (that is, a given node could become a bottleneck if sufficient computation resources are unavailable).
- The runtime system must be capable of dealing with threads which have run a muck.
- It may be impossible to group data such that related items are together. (This experimentation, using benchmarks which are among the worst known in this regard, indicates that this is really not a problem.)

In terms of load balancing, programmers must already deal with the same problem in that shared data may become a bottleneck. As for the second potential objection, that the runtime system must seek out “dangerous” threads, it is possible for threads to be destroyed by holding expired capabilities. For example, freeing all the memory associated with a process could cause a thread executing on that process to be destroyed once it makes a memory access. Finally, the third potential disadvantage (which could be most detrimental to the system) is shown here and in [15, 16] to be very easily avoided. In fact, if no special attention to grouping is paid (other than that already paid by a standard operating system), the system is highly effective.

4 Simulation Methodology

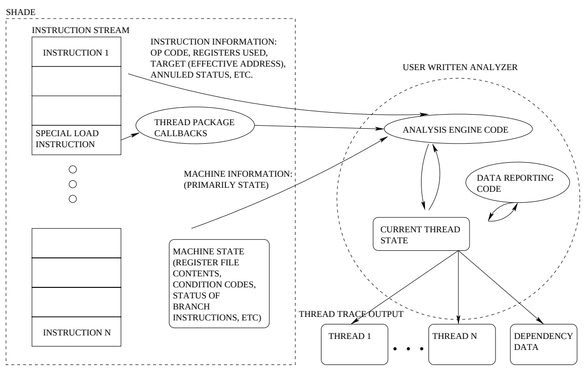

SHADE
INSTRUCTION STREAM
INSTRUCTION 1
SPECIAL LOAD INSTRUCTION
INSTRUCTION N
INSTRUCTION INFORMATION: OP CODE, REGISTERS USED, TARGET (EFFECTIVE ADDRESS), ANNULED STATUS, ETC.
THREAD PACKAGE CALLBACKS
MACHINE INFORMATION: (PRIMARILY STATE)
MACHINE STATE (REGISTER FILE CONTENTS, CONDITION CODES, STATUS OF BRANCH INSTRUCTIONS, ETC)
USER WRITTEN ANALYZER
ANALYSIS ENGINE CODE
DATA REPORTING CODE
CURRENT THREAD STATE
THREAD TRACE OUTPUT
THREAD 1
THREAD N
DEPENDENCY DATA

Figure 5: Shade Simulations

This work uses the Shade simulation suite[21] to extract a program trace from a SPARC binary. Figure 5 shows the simulation mechanism. User written code examines the running instruction stream and records events of interest. Since the Shade suite traces SPARC instructions, the simulated ISA corresponds roughly to that of a typical RISC machine. This obviously does not represent the vector ASAP ISA, however, this work is primarily concerned with the performance of the carpetbag cache which only requires memory access information.

Shade has no knowledge of kernel bound threads. This work uses a custom user level thread package that calls back into the shade analysis code to indicate thread events such as changing threads or blocking on a lock. This information is also used to generate a primitive dependency graph which indicates when one thread must wait for another to finish its computation (and release the lock) before proceeding.

To allow the simulation to be tractable, input sets were restricted to the 250-750 MB range, as appropriate for the particular benchmark. Given that individual PIM sizes are significantly less than even the minimum size presented, many opportunities for movement between nodes are generated. Additionally, simulation was limited to a 32-bit address space. Data sets were divided into three parts: code (as indicated by portions of memory subject to an instruction fetch), the stack (which grows down from the top of the address space), and the heap (everything else). Thread traces were further limited to 500,000,000 instructions.

The shade simulator extracts a trace of each thread along with the dependency information. These traces are then provided to carpetbag cache simulation tools that tack the performance of the cache and the state of the memory system over a homogeneous PIM array. The memory allocation mechanism is inherited from the original shade simulation – specifically, memory is allocated in the order in which it was requested. The data is then placed on PIMs in consecutive chunks corresponding to the given PIM size.

It should be noted that these simulations only examined accesses to the heap. The stack and code demands of very large threads from the same benchmark suite have been shown to be relatively small[15]. Furthermore, since there are other strategies available for dealing with those types of accesses exist (such as loading the code for a given thread on each node before execution, or only transmitting a small top segment of the stack), only the more complicated (and random) heap access is examined.

Although the shade simulation model assumes that the actual program code consists of scalar RISC instructions (which the ASAP handles by choosing an appropriate portion of the vector datapath to execute a given scalar instruction), it should be noted that the vector capabilities of the processor are exploited in terms of reducing the overhead of page translations or cache look-ups. Using a vectorizing compiler, the number of instructions in a given run would be reduced, however, the memory access pattern would not.

5 Benchmarks

This work concentrates on the analysis of a subset of the Data Intensive Systems (DIS) benchmark suite[2, 3] that has been hand threaded. These benchmarks are atypical in that their memory access patterns exhibit a low degree of reuse and non-linear stride. Thus the benchmarks tend to exhibit worst case memory access patterns in that the probability of generating a remote access is higher. Clearly in the case of PIM (and the carpetbag cache in particular) the performance of the memory system is of paramount importance. These "worst case" numbers show the system under stress rather than achieving peak performance. Most benchmark suites, in sharp contrast, are designed to be quickly captured in a processor's cache so as to measure raw computation power. This is somewhat misleading since the performance of most modern architectures is determined by that of the memory system, and in the case of large PIM arrays the numbers hold significantly less meaning.

The research that preceded this work[15, 16] used a single threaded model to investigate the memory access characteristics of the entire benchmark suite. Originally, the work focused on the performance of the SPEC95[18] integer and floating point suites. These experiments yielded simplistic results as the memory access patterns were both regular and easily accommodated by small PIMs. Tests in which the data set sizes were increased did not fare much better in that the benchmarks themselves tend to use data with a high degree of both spatial and temporal locality.

Use of the DIS suite yielded significantly more interesting results, though each of the benchmarks proved very similar in memory system performance. Given the difficulty associated with multi-threading the benchmarks and the high simulation cost, two programs were chosen from the suite as representative of Data Intensive programs.

The first, the Method of Moments (MoM) benchmark represents algorithms which are frequency domain techniques for computing electro-magnetic scattering from complex objects. Typical implementations employ direct linear solves, which are highly computation intensive and can only be applied to reasonably low frequency problems. The faster solvers applied in this benchmark are memory bound since reuse is extremely low and access patterns exhibit non-uniform stride. This benchmark is derived from the Boeing implementation of fast iterative solvers for the Helmholtz equation[6, 8, 7]. This is the most computationally intensive DIS benchmark and it utilizes large matrices with non-uniform stride.

The second, in contrast, is the Data Management (DM) benchmark which implements a simplified object-oriented database with an R-Tree indexing scheme [9, 13]. Three operations are supported: *insert*, *delete*, and *query*. For the purposes of these experiments, only the *query* operation was examined. The indexes are composed of many small objects and have a high degree of pointer chasing.

These two benchmarks exhibit highly non-uniform stride memory accesses with very low reuse, and, given previous experimentation, prove representative of the entire suite. They contrast in the method used to generate that access pattern (large matrices vs. pointer chasing), and in the way in which they are multi-threaded. The Method of Moments benchmark has a very large number of relatively short threads, while the Data Management benchmark has significantly longer threads (though nearly an order of magnitude fewer of them).

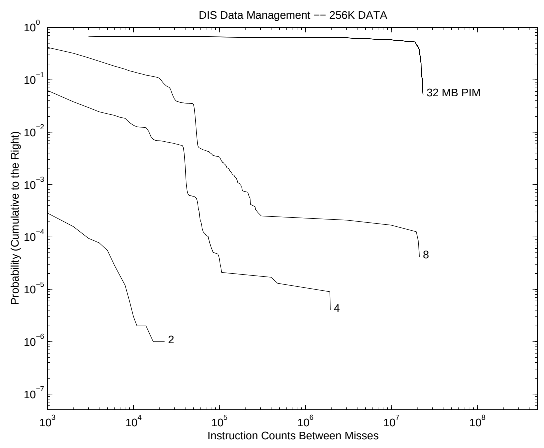
6 Results

The measure of success for the carpetbag cache is presented as the mean run length before an off chip reference is generated. In this case, the run length is the total number of uninterrupted instructions a given thread can execute. Long run lengths (which can amortize the cost of moving the thread) are considered highly desirable. Similarly, the cost associated with achieving these run lengths is measured in terms of the size of the carpetbag cache.

6.1 Baseline Configurations

Previous work has shown that the best performance numbers for these benchmarks are achieved using very large pages [15, 16]. This is not surprising given that they are data intensive and have extremely large working sets. Figure 6 shows the baseline results for a standard VM paging scheme. In particular, it gives the Cumulative Instruction Probability Density (CIPD), which represents the probability that a runlength of size n or greater will be achieved. As the figure clearly shows, it is extremely difficult to achieve run lengths greater than 1000 instructions.

These results account for neither the overhead of translation nor the overhead of fetching the page. However, it is easily observed that a very large number of memory system transactions are generated very quickly. The Carpetbag cache no only generates fewer transactions, but changes round trip communication into pont-to-point communication.



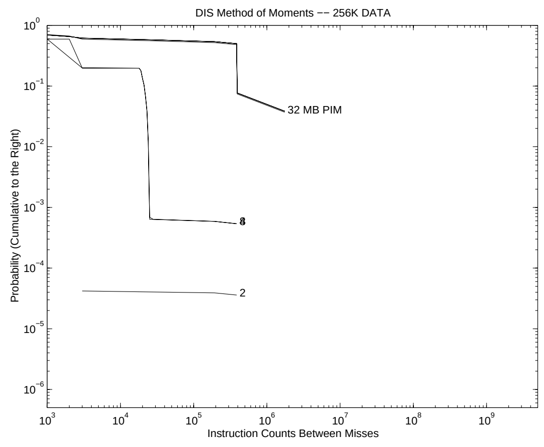


Figure 6: 256 KB Paging VM System

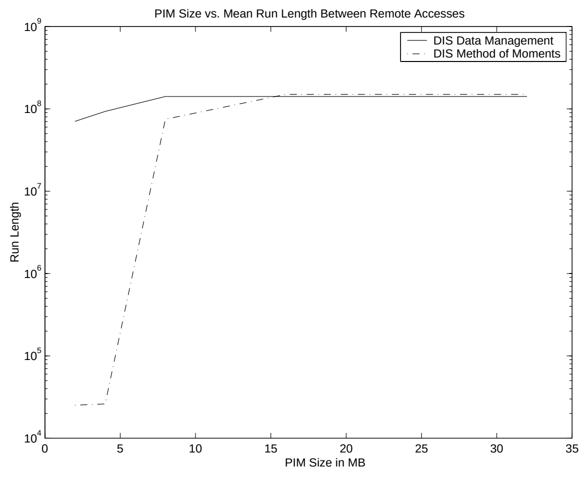
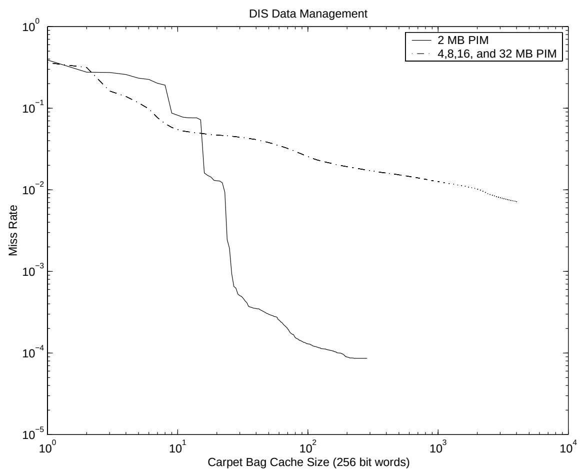


Figure 7: Mean Run Length vs. PIM Size (in MB)



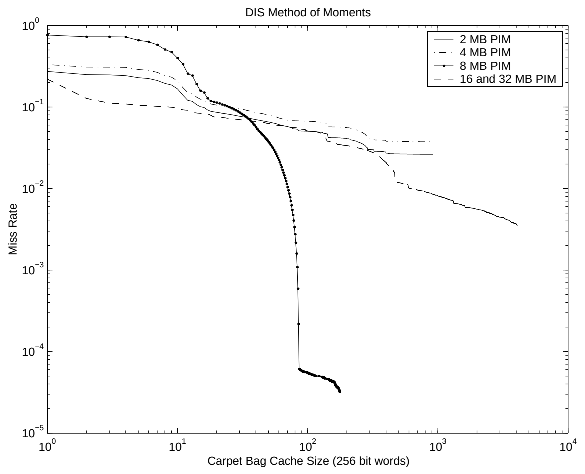


Figure 8: Miss Rate vs. Carpetbag Cache Size

6.2 Mobile Thread

It should be noted that experimentation without the carpetbag cache showed a truly multi-threaded mobile thread model to be unsustainable. The mean run length between off node references (when the thread would move every time a miss was generated and had no carpetbag cache to look back on) never exceeded 500 instructions. This heavy thrashing would have proven detrimental to system performance as the cost of moving the thread could not be amortized over the execution of that thread on a given node.

6.3 Carpetbag Cache

Figure 7 shows that both benchmarks very quickly ramped up to long run lengths between misses. In fact, an 8 MB or greater PIM seems to afford very long runs without remote accesses. This corresponds directly to the size of a PIM which demonstrates significant capture of a working set [15, 16].

Figure 8 shows the effectiveness of the carpetbag cache in terms of size versus miss rate. For either benchmark

$O(10)$ 256-bit words achieve a hit rate near 90%. The only unusual line in either graph is the Method of Moments 8 MB PIM miss rate, which shows a dramatic drop in miss rate with a cache of approximately 100 words. This suggests that an 8 MB partitioning of the data yields a significant performance increase, which is confirmed by hand analysis of the code.

In general, however, 1000 256-bit words (or, approximately 32 KB of data) will yield a less than 1% miss rate. While implementing what effectively amounts to a 1000 entry TLB in hardware (for the carpetbag cache translation) would probably be too costly, trapping to the runtime system and performing a vector enabled look up before generating a high latency remote memory access would not be.

On the ASAP processor, searching for a 32-bit address in a carpetbag is performed very quickly using an oct-tree lookup (the ASAP is capable of operating on 256-bit vectors in 8, 16, or 32-bit chunks). Moving to 64-bit addresses (and assuming the architecture were extended to operate in 64-bit chunks), a quad-tree lookup could be performed. This simple trap to the OS requiring $O(\log_8 1000) \approx 2.08$ or $O(\log_4 1000) \approx 4.98$ iterations through a tight vector loop is a negligible cost to pay in comparison to the latency of a remote network communication.

7 Conclusions and Future Work

This paper presented a unique method of managing continuations through a mobile thread model. This model emphasizes trading bandwidth for latency in that parcel size is considered significantly less important than the number and type of communications required. This is characteristic of the very large parallel arrays we are targeting (and interconnection networks in general).

The data structure enabling this model is the *carpetbag cache* which is designed to capture data on one node for transportation to another. Reasonable performance is achieved with a relatively small cache, whereas, unsurprisingly, the cost increases significantly to achieve hit rates over 90%. These costs, however, are not extraordinarily high when one considers the high cost of communication between nodes. Furthermore, the possibility to balance the cost of communication with the cost of hardware exists in that the performance hit associated with software performing some of the cache's duties can be amortized by avoiding the communication. This leads to the possibility of very powerful hardware/software trade offs. Software further allows more complex trade offs to be made. For example, a node could look at the distance between it and the node to which the thread is moving (and therefore the cost of communication) when deciding how big the cache to be constructed will be. Additionally, these benchmarks represent the worst case numbers because they exhibit (by design) low reuse and low spatial locality. The long run lengths exhibited are largely a product of the caching structure and the nature of the way in which programmers allocate data. Specifically, even though the program may be chasing pointers, the probability of that pointer being on the same PIM node is quite high due to the size of the node and the allocation mechanism. Furthermore, the carpetbag cache caters to the common case – that thrashing will be between two nodes.

Future work will examine the hardware/software trade offs of the carpetbag cache architecture. While this paper demonstrates the viability of the model, several questions as to the specifics of an implementation remain. Specifically, examining the traffic patterns of many fine grain mobile threads traversing a system with realistic communication implementations would prove highly interesting. While for the purposes of these simulations we can assume that such contention is no worse than reading a data hot spot, the specifics of a runtime system implementation will be addressed next. Finally, the impact of a vector ISA on everything from the memory system performance to the ability to quickly resolve a "previous node memory access fault" is of paramount importance to the design of the ASAP ISA.

In addition, a prototype version of a PIM node known as "PIM Lite" is currently in the final stages of design and layout with real silicon expected by the summer of 2001. PIM Lite incorporates many of the key concepts discussed in this paper, such as hardware multi-threading, wide word operations, and a limited thread context cache that can be used to prototype runtime systems supporting carpetbag cache operations.

References

[1] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera System.

[2] Atlantic Aerospace Electronics Corporation. *Data-Intensive Systems Benchmark Suite Analysis and Specification*, 1.0 edition, June 1999.

[3] Atlantic Aerospace Electronics Corporation. *Data Intensive Systems Benchmark Suite*, `http://www.aaec.com/projectweb/dis/`, July 1999.

[4] Jay B. Brockman, Peter M. Kogge, Vincent Freeh, Shannon K. Kuntz, and Thomas Sterling. Microservers: A New Memory Semantics for Massively Parallel Computing. In *ICS*, 1999.

[5] Doug Burger. System-Level Implications of Processor-Memory Integration. *Proceedings of the 24th International Symposium on Computer Architecture*, June, 1997.

[6] B Dembart and E.L. Yip. A 3-d Fast Multipole Method for Electromagnetics with Multiple Levels, December 1994.

[7] M.A. Epton and B Dembart. Low Frequency Multipole Translation for the Helmholtz Equation, August 1994.

[8] M.A. Epton and B Dembart. Multipole Translation Theory for the 3-d Laplace and Helmholtz Equations. *SIAM Journal of Scientific Computing*, 16(4), July 1995.

[9] Guttman. R-Trees: a Dynamic Index Structure for Spatial Searching. In *Proceedings of ACM SIGMOID*, June 1984.

[10] Mary Hall, Peter Kogge, Jeff Koller, Pedro Diniz, Jacqueline Chame, Jeff Draper, Jeff LaCoss, John Granacki, Apoorv Srivastava, William Athas, Jay Brockman, Vincent Freeh, Joonseok Park, and Jaewook Shin. Mapping Irregular Applications to DIVA, A PIM-based Data-Intensive Architecture. In *Supercomputing, Portland, OR*, November 1999.

[11] Peter M. Kogge, Jay B. Brockman, and Vincent Freeh. Processing-In-Memory Based Systems: Performance Evaluation Considerations. In *Workshop on Performance Analysis and its Impact on Design held in conjunction with ISCA, Barcelona, Spain*, June 27-28, 1998.

[12] Peter M. Kogge, Jay B. Brockman, and Vincent W. Freeh. PIM Architectures to Support Petaflops Level Computation in the HTMT Machine. In *3rd International Workshop on Innovative Architectures, Maui High Performance Computer Center, Maui, HI*, November 1-3, 1999.

[13] Banks Kornacker. High-Concurrency Locking in R-Tree. In *Proceedings of 21st International Conference on Very Large Data Bases*, September 1995.

[14] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, and M. Horowitz. Smart memories: A modular reconfigurable architecture. *ISCA*, June 2000.

[15] Richard C. Murphy. *Design Parameters for Distributed PIM Memory Thesis.* MS CSE Thesis, University of Notre Dame, April 2000.

[16] Richard C. Murphy, Peter M. Kogge, and Arun Rodrigues. The Characterization of Data Intensive Memory Workloads on Distributed PIM Systems. In *Porceedings of the Second Workshop on Intelligent Memory Systems, held in conjunction with ASPLOS-IX*, Cambridge, MA November 12-15, 2000.

[17] Notre Dame PIM Development Group. *ASAP Principles of Operation*, February 2000.

[18] SPEC Open Systems Steering Committee. SPEC Run and Reporting Rules for CPU95 Suites. September 11, 1994.

[19] David Patterson, Thomans Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A Case for Intelligent DRAM: IRAM. *IEEE Micro*, April, 1997.

[20] T. Sterling and L. Bergman. A design analysis of a hybrid technology multithreaded architecture for petaflops scale computation. In *International Conference on Supercomputing, Rhodes, Greece*, June 20-25, 1999.

[21] Sun Microsystems. *Introduction to Shade*, June 1997.

[22] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture.* ACM Press, 1992.